Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Julija Petrič

Večkratno razpošiljanje v javi

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Večina današnjih programskih jezikov podpira princip enojnega razpošiljanja, kjer se izvedena metoda določi glede na dinamični tip prejemnika. Možno pa je tudi dvojno in večkratno razpošiljanje, pri čemer se upoštevajo tudi dinamični tipi argumentov metode. V okviru diplomske naloge preštudirajte princip razpošiljanja in programske jezike, ki ga podpirajo. Nato predstavite, kako lahko večkratno razpošiljanje izvedemo v programskem jeziku java, ki le tega ne podpira. Izvedite tudi eksperimentalno ovrednotenje različnih pristopov k izvedbi večkratnega razpošiljanja.

Zahvaljujem se staršem, sestri, bratu in prijateljem za vso podporo pri pisanju diplomske naloge in mentorju za strokovno pomoč in nasvete.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
OOP	object-oriented programming	objektno orientirano programiranje
JVM	Java Virtual Machine	javanski navidezni stroj
BFS	breadth first search	preiskovanje v širino
DFS	depth first search	preiskovanje v globino

Povzetek

Naslov: Večkratno razpošiljanje v javi

Avtor: Julija Petrič

Diplomska naloga predstavlja mehanizme razpošiljanja. Objektno orientirane programske jezike lahko v grobem delimo na jezike, ki omogočajo enojno razpošiljanje in na jezike, ki zagotavljajo večkratno razpošiljanje. Pri enojnem razpošiljanju se izvajalni sistem na podlagi dejanskega tipa objekta prejemnika odloči, katero metodo bo v resnici klical. Vendar ima enojno razpošiljanje nekaj omejitev. Zato čedalje več programskih jezikov omogoča večkratno razpošiljanje, kjer je izbira dejanske klicane metode določena glede na vse argumente klica metode. V diplomski nalogi so predstavljene različne tehnike, kako simulirati večkratno razpošiljanje v programskem jeziku java, ki zagotavlja le enojno razpošiljanje. Naloga zajema tudi enostavne testne programe, s katerimi sem merila hitrost izvajanja posameznih tehnik.

Ključne besede: večkratno razpošiljanje, enojno razpošiljanje, dvojno razpošiljanje.

Abstract

Title: Multiple dispatch in Java

Author: Julija Petrič

In this thesis we talk about dispatch mechanisms. Object oriented programming languages are divided into two main categories: programming languages that supports single dispatch and programming languages that supports multiple dispatch. In single dispatch languages the method to be executed is selected by the dynamic type of the receiver object and static types of method parameters. However, single dispatch has several limitations. Consequently, a growing number of programming languages enables a multiple dispatch, where the method to be executed is selected by the dynamic type of all arguments of method call. Here, different multiple dispatch simulation techniques in java programming language, which otherwise supports only single dispatch, are discussed. We performed simple test programs to evalue the runtime of each individual technique.

Keywords: multiple dispatch, single dispatch, double dispatch.

Poglavje 1

Uvod

Razpošiljanje je mehanizem, ki na podlagi argumentov klica metode določi katera metoda bo klicana. V grobem lahko razpošiljanje delimo na statično in dinamično razpošiljanje. Pri statičnem razpošiljanju se klic metode razreši že v času prevajanja programa, glede na statične tipe argumentov. Dinamično razpošiljanje pa omogoča, da se izvajalni sistem na podlagi dejanskih tipov argumentov odloči, katero metodo bo dejansko poklical. Dinamično razpošiljanje omogoča polimorfizem, ki je ena glavnih lastnosti objektno orientiranih programskih jezikov.

Dinamično razpošiljanje lahko delimo glede na število argumentov, na podlagi katerih se izvajalni sistem odloči, katero metodo bo poklical:

- enojno razpošiljanje,
- dvojno razpošiljanje,
- večkratno razpošiljanje.

Jeziki kot so Java, C++, Smalltalk, itd. podpirajo mehanizem enojnega razpošiljanja. Izbira metode, ki bo dejansko klicana je določena glede na en dejanski tip argumentov. Ponavadi je to objekt prejemnik.

Kadar je izbira metode določena glede na dejanske tipe vseh argumentov, pravimo takšnemu mehanizmu večkratno razpošiljanje. Zagotavljajo ga programski jeziki kot CLOS, Dylan, Julia, Multijava, itd.

Ponavadi implementiramo mehanizem dvojnega in večkratnega razpošiljanja v programskih jezikih z enojnim razpošiljanjem, da se izognemo omejitvam enojnega razpošiljanja. Pri dvojnem razpošiljanju je dejanska klicana metoda izbrana glede na dva dejanska tipa argumentov.

1.1 Motivacija in cilji

Razumevanje tega, kaj se dogaja v ozadju med prevajanjem in izvajanjem programa, je podlaga za pisanje dobrega, zanesljivega programa. Tekom let smo ugotovili, da imajo nekateri programerji s tem težavo. Z diplomsko nalogo želim predstaviti mehanizem razpošiljanja, ki ga uporabljata tako prevajalnik kot tudi izvajalni sistem. Predstavitev teh mehanizmov bo programerje pripeljala do boljšega razumevanja, kako se v programskih jezikih razrešujejo klici metod.

Cilj diplomske naloge je podrobnejša opredelitev ter predstavitev mehanizmov razpošiljanja. Najprej se osredotočimo na enojno razpošiljanje in na omejitve, ki nastanejo v programskih jezikih, ki podpirajo le ta mehanizem. Nato pa preidemo na dvojno in večkratno razpošiljanje. Podrobneje si tudi pogledamo, kako je enojno razpošiljanje implementirano v programskem jeziku java in s kakšnimi pristopi lahko simuliramo mehanizem dvojnega in večkratnega razpošiljanja. Prikazana je tudi časovna zahtevnost različnih mehanizmov in pristopov.

1.2 Pregled vsebine

V poglavju 2 se osredotočimo na predstavitev mehanizmov. Najprej predstavimo enojno razpošiljanje in omejitve, ki nastajajo pri tem mehanizmu. Nato pa preidemo na dvojno in večkratno razpošiljanje. Pri dvojnem razpošiljanju si na hitro pogledamo, kako ga lahko implementiramo v programskem jeziku kot je na primer java.

Poglavje 3 je namenjeno predstavitvi, kako v programskem jeziku java

razrešujemo klice metod. Najprej si pogledamo, kako prevajalnik razrešuje klic, nato pa še, kako izvajalni sistem reši klic metode. Pogledamo si tudi glavne prednosti in lastnosti jezika java. Poglavje vsebuje opise različnih pristopov, kako lahko simuliramo dvojno in večkratno razpošiljanje. Pogledamo si glavne prednosti in slabosti posameznih pristopov.

V poglavju 4 pa naredimo še enostavne testne programe, s katerimi izmerimo hitrost izvajanja posameznih pristopov in jih nato primerjamo.

Poglavje 2

Mehanizem razpošiljanja

Vsak programski jezik mora imeti implementiran nek mehanizem, s katerim razrešuje klice metod. Takšnemu mehanizmu pravimo razpošiljanje (*angl. dispatch*).

V grobem lahko razpošiljanje delimo na dve vrsti:

- statično razpošiljanje,
- dinamično razpošiljanje.

Pri statičnem razpošiljanju se izbira metode določi med prevajanjem programa. Uporablja se, kadar imamo več metod definiranih z enakim imenom in različnimi parametri (*angl. method overloading*). Prednost statičnega razpošiljanja je, da je čas izvajanja programa krajši, saj se izbira metode določi med prevajanjem programa. Slabost je v tem, da ni fleksibilen, saj ne moremo vedno določiti, katero metodo bomo klicali, ker ne poznamo še dejanskega tipa objekta.

Spodnji primer 2.1 prikazuje enostavni java program, kjer uporabnik vnese številko, program pa nam nato izpiše, kateri tip objekta je bil uporabljen. Prevajalnik med prevajanjem programa ne ve, katera metoda *getClass()* bo dejansko klicana med izvajanjem programa. Zato potrebujemo dinamično razpošiljanje.

Program 2.1: Izpis tipa številke

```
class ObjektStevilke {
    public static void main(String [] args) {
       Number n = NumberFormat.getInstance().
                                  parse(args[0]);
       System.out.println(n.getClass());
    }
}
```

Zaradi dinamičnega razpošiljanja nam bo program 2.1 vrnil različne rezultate.
V primeru, ko bomo vnesli celo število:
class java.lang.Long
Če vnesemo decimalno število, pa nam vrne:
class java.lang.Double

Dinamično razpošiljanje omogoča polimorfizem (*angl. polymorphism*). Polimorfizem je eden najpomembnejših gradnikov objektno orientiranih programskih jezikov. Polimorfizem bi lahko definirali kot princip, da lahko več objektov razume isto sporočilo, vendar nanj odgovorijo vsak na svoj način [23].

Dinamično razpošiljanje se uporablja, kadar imamo virtualne metode (*angl. virtual method*) ali povožene metode (*angl. method overriding*). Izbira metode ni določena med prevajanjem programa, zato je čas izvajanja programa daljši kot pri statičnem razpošiljanju.

Da lahko zagotovimo fleksibilnost objektov med izvajanjem programa, objektno orientirani programski jeziki uporabljajo dinamično razpošiljanje. Objektno orientirane programske jezike lahko ločimo v dve glavni skupini, glede na število argumentov metode, ki sodelujejo pri razpošiljanju. Enojno razpošiljanje omogočajo OOP programski jeziki, kjer je izbira metode določena glede na en argument, pri večkratnem razpošiljanju pa je izbira metode določena glede na več argumentov posredovanih ob klicu metode, ponavadi pridejo v poštev vsi [6].

Ker imajo objektno orientirani programski jeziki z enojnim razpošiljanjem veliko omejitev, pogosto implementiramo dvojno razpošiljanje in s tem rešimo širši krog problemov, kot je na primer problem binarnih metod. Pri dvojnem razpošiljanju je izbira metode določena glede na dva argumenta metode.

2.1 Enojno razpošiljanje

Programski jeziki kot so java, C++, smalltalk, itd., podpirajo mehanizem enojnega razpošiljanja. To je mehanizem, ki se na podlagi dinamičnega tipa enega argumenta klica metode med izvajanjem programa odloči, katera metoda bo dejansko poklicana. Ostali argumenti, ki so posredovani ob klicu metode, to so na primer argumenti metode, ne vplivajo pri izbiri metode.

V programskem jeziku java, objekt *prejemnik* določa, katera metoda bo dejansko poklicana in izvedena. Objekt prejemnik je objekt "pred piko". Sintaksa klica metode v javi izgleda: *prejemnik.metoda(*$argument_1$*,* $argument_2$*, ...)* [22].

Za primer vzemimo spodnjo enostavno razredno hierarhijo (program 2.2). Imamo razred *Zival*, ki predstavlja nadrazred podrazredov *Macka* in *Pes*. Razredi vsebujejo metodo *oglasanje()*, kjer podrazreda povozita metodo nadrazreda.

Program 2.2: Razredna hierarhija

```
class Zival {
    void oglasanje(Zival z) {
        System.out.println("Oglasanje zivali.");
    }
}
class Macka extends Zival {
    @Override
    void oglasanje(Zival m) {
        System.out.println("Macka mijavka.");
    }
}
```

```
class Pes extends Zival {
    @Override
    void oglasanje(Zival p) {
        System.out.println("Pes␣laja.");
    }
}
```

V testnem razredu (program 2.3) nato ustvarimo dva nova objekta *Macka* ter *Pes* in pokličemo metodo *oglasanje()*.

Program 2.3: Testni program

```
class Test {
    public static void main(String[] args) {
        Zival macka = new Macka();
        Zival pes = new Pes();
        macka.oglasanje(pes); // izpise "Macka mijavka."
    }
}
```

S pomočjo mehanizma enojnega razpošiljanja, izvajalni sistem vsak trenutek ve, kakšen je dejanski tip objekta *macka*, čeprav smo mu dodelili splošnejši tip, *Zival*. Na podlagi dejanskega tipa se java med izvajanjem programa odloči, katera metoda *oglasanje()* bo dejansko klicana [18].

V nekaterih OOP jezikih, kot sta na primer java in C++ lahko implementiramo metode z enakim imenom, a različnimi tipi parametrov. Takšne metode lahko vračajo tudi različne rezultate.

Program 2.4: *Razširitev razreda Macka iz 2.2*

```
public class Macka extends Zival {
    /** enako kot zgoraj**/
    void oglasanje(Pes p) {
        System.out.println("Pes␣laja␣na␣macko.");}
}
```

Sedaj imamo razširjen razred *Macka* z novo metodo *oglasanje(pes)*. Ko iz zgornjega testnega razreda (program 2.3) kličemo metodo *oglasanje()*, je dejansko klicana metoda, še vedno metoda *"Macka mijavka."*. To je posledica in omejitev enojnega razpošiljanja, saj se, kot je že omenjeno, izbira metode določi glede na objekt prejemnik. Ostali argumenti, posredovani ob klicu metode ne igrajo nobene vloge, saj so statično dodeljeni že v času prevajanja programa.

2.2 Problem enojnega razpošiljanja

Mehanizem enojnega razpošiljanja je uporaben, kadar nam zadošča, da na podlagi dinamičnega tipa objekta prejemnika pokličemo želeno metodo. Vendar, kot lahko vidimo že v primeru 2.4, ima enojno razpošiljanje precej omejitev. V nadaljevanju si bomo pogledali problem binarnih metod.

Obstajajo tudi drugi znani problemi kot na primer problem razširjanja objektne hierarhije in dodajanja novih operacij, ki delujejo nad to objektno hierarhijo. Več o tem si lahko preberete v [3].

2.2.1 Problem binarnih metod

Splošno znan problem v objektno orientiranih programskih jezikih je problem, kjer za izbiro želene metode potrebujemo več kot en dinamični tip argumentov [3]. Poglejmo si zgornji primer (program 2.4), kjer potrebujemo dinamični tip objekta prejemnika (v našem primeru *Macka*) in dinamični tip objekta parametra (*Pes*), da zagotovimo izpis na ekranu *"Pes laja na macko."*.

Binarne metode so metode, ki delujejo na dveh ali več objektov sorodnega tipa. Pri programiranju velikokrat zasledimo uporabo binarnih metod, primer je že metoda iz poglavja 2.1, *Zival.oglasanje(Zival)*. Pogosti primeri uporabe so še aritmetične operacije (seštevanje, odštevanje, množenje, deljenje, itd.) in primerjava objektov (kot na primer primerjava enakosti *Object.equals(Object)*) [21].

Povoziti binarne metode v podrazredih ni primerno, saj to krši standardno kovariantno (*angl. covariant*) pravilo za preverjanje tipov funkcij [1]. V zgornjem primeru (program 2.2) ni primerno povoziti metodo *oglasanje(Zival)* v razredu *Zival* z metodo *oglasanje(Macka)* v razredu *Macka*. Namreč v primeru, ko bi prišli do klica metode *z1.oglasanje(z2)*, kjer je dinamičen tip spremenljivke *z1*, *Macka*, bi v programskih jezikh z enojnem razpošiljanju klicali metodo iz razreda *Macka*. Vendar ne smemo pozabiti na primer, kjer spremenljivka *z2* ni tipa *Macka* (na primer, je tipa *Zival*). V tem primeru pride do napake med izvajanjem (*angl. run time error*) [22].

V programskem jeziku java ne pridemo do napake, saj se takšne metode smatrajo kot statične večkrat definirane metode (*angl. statically overloading methods*). Čeprav nam java ne vrne napake med izvajanjem, še vedno program ne vrne želenih rezultatov, kot lahko vidimo iz programa 2.4. S klicem metode *macka.oglasanje(pes)*, kjer imata oba objekta *macka* in *pes* statični tip *zival*, bo dejansko klicana metoda, še vedno metoda, ki nam izpiše *"Macka mijavka."* in ne *"Pes laja na macko."*, kot bi to želeli.

Takšnemu problemu pravimo tudi problem binarnih metod. Izognemo se mu z implementacijo mehanizma dvojnega razpošiljanja.

2.3 Dvojno razpošiljanje

Mehanizem dvojnega razpošiljanja je posebna verzija mehanizma večkratnega razpošiljanja, kjer prideta v poštev tako objekt prejemnik, kot tudi parameter metode. V glavnem takšen mehanizem implementira vsak programer zase (ni vgrajena funkcionalnost nobenega programskega jezika).

Dvojno razpošiljanje ponavadi implementiramo kot zaporedje enojnih razpoši-ljanj. Program 2.5 prikazuje implementacijo metode *oglasanje()* z uporabo dvojnega razpošiljanja.

Program 2.5: Simulacija dvojnega razpošiljanja

```
class Zival {
        void oglasanje(Zival z) {
```

```
        System.out.println("glava");
    }
    void oglasanjeMacke(Macka m) {
        System.out.println("glava␣m.");
    }
    void oglasanjePsa(Pes p) {
        System.out.println("glava␣p.");
    }
}
class Macka extends Zival {
    void oglasanje(Zival z) {
         z.oglasanjeMacke(this);
    }
    void oglasanjeMacke(Macka m) {
        System.out.println("Macka␣mijavka.");
    }
    void oglasanjePsa(Pes p) {
        System.out.println("Macka␣piha␣na␣psa.");
    }
}
class Pes extends Zival {
    void oglasanje(Zival z) {
         z.oglasanjePsa(this);
    }
    void oglasanjeMacke(Macka m) {
        System.out.println("Pes␣laja␣na␣macko.");
    }
    void oglasanjePsa(Pes p) {
        System.out.println("Pes␣laja");
    }
}
```

Klic prve metode *"oglasanje()"* predstavlja prvo pošiljanje, klic druge metode *"oglasanjeX()"* pa predstavlja drugo pošiljanje. Torej se argumenti

premešajo tako, da je vsak poslan enkrat.

Metode, ki so namenjene klicanju tudi izven razreda so metode *"oglasanje()"*, metode *"oglasanjeX()"* pa so namenjene dejanski implementaciji in jih ne kličemo iz zunanjih razredov.

Poglejmo si, kakšne rezultate dobimo, ko kličemo metode z različnimi tipi argumentov. Sedaj je dejanska klicana metoda odvisna od vseh dejanskih tipov argumentov. V primeru, ko kličemo *macka.oglasanje(pes)*, kjer sta *macka* in *pes* statičnega tipa *zival*, se nam izpiše *"Pes laja na macko."*. Sprva se nam lahko zdi rezultat nenavaden, saj pričakujemo, da bo dejanska klicana metoda, neka metoda iz razreda *Macka*. Vendar, če podrobnejše pogledamo programsko kodo 2.5, lahko vidimo, da je najprej dejanska klicana metoda *oglasanje()* iz razreda *Macka*, nato pa iz te metode kličemo še *z.oglasanjeMacke(this)*, kjer je *z* dejanskega tipa *Pes* in *this* predstavlja objekt *Macka*. Sedaj naš klic metode izgleda *Pes.oglasanjeMacke(Macka)*, zato začnemo iskati metodo v razredu *Pes* in ko jo najdemo, dobimo rezultat *"Pes laja na macko."*.

Enake rezultate dobimo tudi, če v testnem programu spremenljivkam *macka* in *pes* ne dodelimo splošni tip *Zival*, vendar njun dejanski tip (*Macka macka = new Macka() in Pes pes = new Pes()*).

V spodnjem programu 2.6 si lahko pogledamo še druge kombinacije izpisa metod z različnimi tipi argumentov.

Program 2.6: Testni program

```
public static void main(String[] args) {
    Zival macka = new Macka();
    Zival pes = new Pes();
    Pes p = new Pes();
    Macka m = new Macka();
    macka.oglasanje(pes);    // Pes laja na macko.
    macka.oglasanje(p);      // Pes laja na macko.
    pes.oglasanje(macka);    // Macka piha na psa.
    pes.oglasanje(m);        // Macka piha na psa.
    macka.oglasanje(macka); // Macka mijavka.
```

```
    pes.oglasanje(pes);      // Pes laja.
}
```

Mehanizem dvojnega razpošiljanja lahko v programskem jeziku java simuliramo tudi s pomočjo preverjanja tipa objekta z *instanceof* in *if-else* stavkov (program 2.7).

Program 2.7: Simulacija s preverjanjem tipov

```
class Macka extends Zival {
    void oglasanje(Zival z) {
        if(z instanceof Macka)
            oglasanjeMacke((Macka)z);
        else if(z instanceof Pes)
            oglasanjePsa((Pes)z);
        else
            //NAPAKA
    }
    void oglasanjeMacke(Macka m) {...}
    void oglasanjePsa(Pes p) {...}
}
class Pes extends Zival {
    void oglasanje(Zival z) {
        if(z instanceof Macka)
            oglasanjeMacke((Macka)z);
        else if(z instanceof Pes)
            oglasanjePsa((Pes)z);
        else
            //NAPAKA
    }
    void oglasanjeMacke(Macka m) {...}
    void oglasanjePsa(Pes p) {...}
}
```

Simulacija dvojnega razpošiljanja ima nekaj slabosti. Prvič preverjanje

tipov (*angl. type-casting*) z *if-else* stavki, kot lahko vidimo v zgornjem primeru (program 2.7), vzame veliko časa za pisanje in lahko vsebuje veliko napak. Drugič, imamo dodatno obremenitev, saj se sklicujemo na drugo metodo. In tretjič, razširitev programa z dodajanjem novega dogodka bi zahtevala dodaten *if-else* stavek [26].

Največkrat zasledimo simulacijo dvojega razpošiljanja z implementacijo načrtovalskega vzorca *obiskovalec*. V poglavju 3.4.1 si bomo podrobno pogledali implementacijo takšnega pristopa.

2.4 Večkratno razpošiljanje

O mehanizmu večkratnega razpošiljanja govorimo, kadar je izbira metode določena glede na dinamične tipe vseh argumentov, posredovanih pri klicu metode [21].

Večkratno razpošiljanje se največkrat pojavi v programskih jezikih, ki so dinamično tipizirani, ker je preverjanje tipov (*angl. type checking*) izvedeno med izvajanjem programa, torej je lahko izbira metode optimalno izbrana glede na specifičen klic metode [26].

Programski jeziki, ki podpirajo zgoraj omenjeni mehanizem so: CLOS, ki je tudi eden izmed prvih jezikov, ki je podpiral večkratno razpošiljanje, poleg tega pa še: Dylan, Cecil, Julia, itd.

V OOP jezikih, ki so statično tipizirani, večkratno razpošiljanje predstavlja nekaj ključnih omejitev, zato ni popularno v teh jezikih [26]. V nadaljevanju diplomske naloge, si bomo pogledali načine, kako lahko implementiramo mehanizem večkratnega razpošiljanja v programskem jeziku java, ki je statično tipiziran jezik.

Večkratno razpošiljanje je fleksibilnejše in omogoča večji krog operacij kot enojno razpošiljanje. Vsak problem enojnega razpošiljanja lahko rešimo s programskimi jeziki, ki zagotavljajo večkratno razpošiljanje. Po drugi stani pa bi bila simulacija večkratnega razpošiljanja v programskih jezikih, ki zagotavljajo samo enojno razpošiljanje, zahtevnejša in bi zanjo potrebovali veliko

vrstic kode [21].

Glavne razlike mehanizmov se poznajo že pri implementaciji metod, saj običajno metode v programskih jezikih z večkratnim razpošiljanjem, niso definirane v razredih, kot je običajno za metode v jezikih z mehanizmom enojnega razpošiljanja, temveč jih lahko definiramo posebej, izven razredne hierarhije.

Poznamo simetrično in asimetrično večkratno razpošiljanje. Razpošiljanje je simetrično, če so vsi argumenti metode enakovredni. Asimetrično večkratno razpošiljanje ponavadi uporabi leksikografski vrstni red, kjer so začetni argumenti pomembnejši kot naslednji. Simetrično večkratno razpošiljanje je bolj intuitivno in je manj nagnjeno k napakam [2].

Spodnji primer 2.8 je napisan v programskem jeziku nice, ki omogoča implementacijo metod izven razredov. Primer je namenjen predstaviti, kako izgledajo metode definirane izven razredov. Prva metoda *oglasanje()* je namenjena deklaraciji metode, šele nato implementiramo metode za posamezne razrede. Programski jezik nice je bil razvit iz jezika java in je primer jezika, ki rešuje problem binarnih metod z večkratnim razpošiljanjem. Ker lahko metode implementiramo izven razredov in jih lahko dodajamo brez potrebe po spreminjanju programske kode razredov, to rešuje tudi problem razširjanja objektne hierarhije in razširjanje operacij nad to objektno hierarhijo.

Program 2.8: Primer deklaracije metod v jeziku Nice

```
class Zival {}
class Macka extends Zival {}
class Pes extends Zival {}

//deklaracija metode
void oglasanje(Zival zival);

//prevzeta implementacija
oglasanje(zival) {
    println("Oglasanje␣zivali.");
}
```

```
//metoda za razred Macka
void oglasanje(Macka m) {
    println("Macka␣mijavka.");
}

//metoda za razred Pes
void oglasanje(Pes p) {
    println("Pes␣laja.");
}
```

2.5 Programski jeziki

V tem poglavju si bomo pogledali programske jezike, ki omogočajo mehanizem večkratnega razpošiljanja. Najprej si bomo pogledali programski jezik CLOS, ki je bil prvi jezik, ki je omogočal večkratno razpošiljanje. Pogledali si bomo še jezike julia in dylan. Julia je bila izbrana kot primer relativno novega programskega jezika, ki podpira večkratno razpošiljanje že od samega začetka. Dylan je programski jezik razvit iz jezika CLOS in ima podoben mehanizem razpošiljanja.

2.5.1 CLOS

Common Lisp Object System ali CLOS je objektno orientirana različica jezika Common Lisp. Je moderen, multi-paradigm, visokozmogljiv, ANSI-standardiziran (American National Standards Institute) in najbolj dodelan programski jezik iz družine jezikov Lisp. Znan je kot zelo fleksibilen, z odlično podporo objektno orientiranega programiranja in s hitrimi prototipnimi zmogljivosti. Odličen je za razvoj aplikacij na strežniški strani [8].

Osnovna ideja v programskem jeziku CLOS je, da generično funkcijo sestavlja več metod. Metode so lahko implementirane za različne kombinacije objektov. V času izvajanja programa bo dejanska klicana metoda izbrana

glede na kateri koli argument generične funkcije ali pa glede na vse argumente. Program 2.9 prikazuje enostavno razredno hierarhijo in definicijo metod [22].Več o OOP jeziku CLOS si lahko preberete v [5], [15] in [25].

Program 2.9: Primer definicije razredov in metod v jeziku CLOS

```
//definicija razredov
(defclass zival () ())
(defclass macka (zival) ())
(defclass pes (zival) ())

//definicija metod
(defmethod oglasanje ((x zival))
    ;; implementacija za razred zival
)
(defmethod oglasanje ((x macka))
    ;; implementacija za razred macka
)
(defmethod oglasanje ((x pes))
    ;; implementacija za razred pes
)
```

2.5.2 Julia

Julia je visoko nivojski in zmogljiv dinamični programski jezik za numerično programiranje. Zagotavlja napreden prevajalnik JIT (*just-in-time*), ki v kombinaciji z obliko programskega jezika pripomore k visoki zmogljivosti programov, omogoča porazdeljeno vzporedno izvedbo programa, visoko numerično natančnost in vsebuje obsežno knjižnico matematičnih funkcij. Knjižnica *Base* jezika julia, je že v veliki meri napisana v samem programskem jeziku Julia, združuje pa tudi odprto-kodne knjižnice programskega jezika C in fortran za linearno algebro, generiranje naključnih števil, procesiranje signalov ter obdelavo nizov [13].

Programski jezik Julia podpira večkratno razpošiljanje, kar je zelo praktično pri delu z matematičnimi operacijami. Matematične operacije so namreč odvisne od vseh argumentov (na primer, operacija $x + y$ je odvisna od x in y) [14].

Tipe parametrov lahko v funkcijah omejimo z dvojnim podpičjem **::** (glej program 2.10). Če tipe parametrov eksplicitno ne omejimo, so splošnega tipa.

Program 2.10: Primer metode v jeziku julia

```
julia> f(x::Float64, y::Float64) = x + y
f (generic function with 1 method)
```

Zgornjo funkcijo f lahko kličemo samo v primeru, ko sta x in y tipa *Float64*, v nasprotnem primeru nam program javi napako, kot je to prikazano v 2.11.

Program 2.11: Primer klica metode v jeziku julia

```
julia> f(1.0,1.0)
2.0
julia> f(1,1)
ERROR: MethodError: ‘f‘ has no method matching
                                  f(::Int64, ::Int64)
```

Funkcijo f lahko enostavno definiramo še za ostale tipe parametrov, kot na primer za tip *Int64* ali pa želimo omejiti na splošen tip, ki predstavlja vsa števila *Number*. Primer definicije je predstavljen v programu 2.12.

Program 2.12: Različne definicije metod

```
julia> f(x::Int64, y::Int64) = x - y
f (generic function with 2 methods)
julia> f(x::Number, y::Number) = x * y
f (generic function with 3 methods)
```

Kadar sta oba parametra *Float64*, je klicana funkcija *f(Float64, Float64)*, če pa sta oba parametra tipa *Int64*, je klicana funkcija *f(Int64, Int64)*. V pri-

meru, ko vnesemo poljubni tip števila (na primer *f(Float64, Int64)*) kličemo splošno funkcijo *f(Number, Number)*. Poglejmo si, kakšne izpise dobimo za različne tipe parametrov (program 2.13):

Program 2.13: Izpisi metod

```
julia> f(1.0, 2)
2.0
julia> f(4.0, 2)
8.0
julia> f(5,5)
0
julia> f(10,10.0)
100.0
julia> f(2.5,2.5)
5.0
```

2.5.3 Dylan

Programski jezik dylan je razvilo podjetje Apple Cambridge, kot del Appleove napredne tehnološke skupine (*angl. Apple's Advenced Tehnology Group*). Njihov cilj je bil, da razvijejo nov sistemski programski jezik za razvijanje aplikacij za Apple Newton PDA.

V jeziku dylan so znaki (*angl. characters*), nizi (*angl. strings*), števila, tabele, vektorji, vrednost resnice ali neresnice (*angl. boolean*), metode, generične funkcije in razredi predstavljeni kot objekti.

Tipa spremenljivke ni potrebno določiti. Če pa tip spremenljivke določimo, potem se pravilnost tipa preveri med prevajanjem programa (statično preverjanje). Če tipa spremenljivke ne določimo, se preverjanje pravilnosti zgodi med izvajanjem programa (dinamično preverjanje).

Večkratno razpošiljanje je ena najmočnejših lastnosti v programskem jeziku dylan. Metode so deklarirane izven razredov, metoda, ki bo izvedena v trenutku klica, pa je izbrana glede na vse parametre [4], [11].

Primer, kako ustvarimo razredno hierarhijo v programskem jeziku dylan in kako definiramo metode za te razrede, je prikazan v spodnjem programu 2.14:

Program 2.14: Primer definicije razredov in metod v jeziku dylan

```
//definicija razredne hierarhije
define class <zival> (<object>)
    ...
end class <zival>;
define class <macka> (<zival>)
    ...
end class <macka>;
define class <pes> (<zival>)
    ...
end class <pes>;

//definicija metod
define generic oglasanje-zival (z :: <zival>) => ();

define method oglasanje-zival (z :: <zival>) => ()
  //implementacija za razred zival
end;

define method oglasanje-zival (m :: <macka>) => ()
  //implementacija za razred macka
end;

define method oglasanje-zival (p :: <pes>) => ()
  //implementacija za razred pes
end;
```

2.5.4 Ostali programski jeziki

Obstajajo še drugi programski jeziki, ki zagotavljajo množično pošiljanje. Cecil je prototipno orientirani jezik, ki podpira večkratno razpošiljanje, dinamično dedovanje in statično preverjanje tipov [22].

V programskem jeziku nice lahko metode definiramo izven razredov. To pomeni, da lahko dodajamo metode k obstoječemu razredu, brez spreminjanja njegove kode. Izbira klicane metode je določena med izvajanjem programa glede na vse parametre [21].

Multijava je razširitev programskega jezika java z odprtimi razredi (*angl. open class*) in mehanizmom večkratnega razpošiljanja. Novo metodo lahko dodamo k obstoječemu razredu brez urejanja tega razreda oziroma, sploh ne potrebujemo njegove izvorne kode. S pomočjo mehanizma večkratnega razpošiljanja je izbira metode odvisna od vseh dinamičnih tipov argumentov klica metode in ne samo od objekta prejemnika [20].

Poglavje 3

Razpošiljanje v programskem jeziku Java

3.1 Programski jezik java

Java je visoko nivojski objektno orientiran programski jezik, ki so ga leta 1991 pričeli in leta 1995 končali razvijati v podjetju Sun Microsystems v okviru projekta Oak, kot zamenjavo oz. preprostejšo obliko programskega jezika C++.

Metode v jeziku java imajo privzeto vrednost *virtual*. Če želimo, da se klic metode razreši že v času prevajanja programa, moramo metodo označiti kot *final* oz. *static*. Če metoda nima takih oznak, izvajalni sistem izbere, katero metodo bo dejansko klical, glede na dejanski tip objekta prejemnika. Temu mehanizmu pravimo enojno razpošiljanje.

Programski jezik java je statično tipiziran jezik, kar pomeni, da prevajalnik preveri, če so vsi določeni parametri in spremenljivke pravega tipa, ki jim je bil dodeljen.

Java program se običajno prevede v niz ukazov javanske vmesne kode in binarni format (datoteka *.class*), določen v specifikacijah javanskega navideznega stroja (*angl. java virtual machine, JVM*) [9].

3.2 Klic metod v javi

Razumevanje programskega jezika java je zelo pomembno, zato si moramo najprej podrobneje pogledati, kako se dejansko ugotovi klicana metoda ter postopek razreševanja klica.

V prvem koraku prevajalnik na podlagi imena metode in statičnega tipa objekta prejemnika določi razrede ali vmesnike, kjer bo iskal metode s takšnim imenom.

V naslednjem koraku poišče vse možne metode, ki so dostopne (*angl. accessible*) in primerne (*angl. applicable*) glede na ime metode in njenih argumentov. To so metode, kjer bo klic za dane argumente uspel.

V primeru, ko je takšnih metod več, prevajalnik izbere najbolj ustrezno (*angl. most specific*). Podpis (podpis metode je ime metode ter število in tipi njenih parametrov) najbolj ustrezne metode in tip, ki ga vrne ta metoda (*angl. return type*), bosta uporabljena pri dinamičnem razpošiljanju v času izvajanja [9].

Metoda je *dostopna*, če je v obsegu vidnosti objekta prejemnika. Na primer privatne metode so vidne samo v razredu, kjer so definirane. Več o tem, kdaj je metoda *primerna* in o iskanju najbolj ustrezne metode, je zapisno v specifikacijah jezika java [9] ter v specifikacijah JVM [16].

V zadnjem koraku prevajalnik zaključi iskanje metode, če gre za konstruktor, ali pa je metoda opremljena s *static*, *private* ali *final*. V vmesno kodo se zapiše natančen klic metode in postopek razreševanja je zaključen. Ker je željena metoda najdena že v postopku prevajanja, takšnemu razreševanju pravimo tudi statično razpošiljanje (*angl. static dispatch*).

Statično razpošiljanje ni možno v vseh primerih kot smo lahko videli v poglavju 2.1, zato prevajalnik prepusti iskanje prave metode izvajalnemu sistemu in v vmesno kodo se zapiše navidezni klic (*angl. virtual method call*).

Ko naletimo na navidezni klic, izvajalni sistem preveri dejanski tip spremenljivke in poskuša najti ustrezno metodo. Najprej išče metodo v razredu dejanskega tipa spremenljivke, če je ne najde, preišče še vse njegove nadrazrede.

Da lahko izvajalni sistem zagotovi učinkovitost iskanja dejanske klicane metode, izdela tabelo navideznih metod (*angl. virtual method table*). Brez seznama te tabele, bi bilo iskanje neučinkovito, saj mora pogosto preiskati celotno razredno hierarhijo [18].

3.3 Večkratno razpošiljanje

Tehnik, kako simulirati mehanizem večkratnega razpošiljanja v programskem jeziku, kot je na primer java, je več. Lahko uporabimo vzorce načrtovanja (*angl. design pattern*), preverjamo dinamične tipe objektov ali uporabimo mehanizem odsevnosti (*angl. reflection*) [19].

V nadaljevanju si bomo najprej pogledali simulacijo večkratnega razpošiljanja z uporabo pristopa preverjanja tipov, nato pa še z uporabo mehanizma odsevnosti.

Pristopi so razloženi z uporabo spodnje razredne hierarhije (program 3.2), kjer nad to hierarhijo izvajamo poljubne operacije.

Program 3.1: Razredna hierarhija

```
interface Test {}
    class A implements Test {}
        class B extends A {}
            class C extends B {}
```

3.3.1 Preverjanje tipov

Lažji, programersko manj zahtevnejši pristop simulacije mehanizma večkratnega razpošiljanja, je pristop s pomočjo preverjanja tipov objektov. V programskem jeziku Java tipe preverjamo z že vgrajenim operatorjem *instanceof* (primer preverjanja prikazan v programu 3.2) [19].

Program 3.2: Primer preverjanja z *instanceof*

```
if(objekt1 instanceof razred)
```

Operator *instanceof* vrne resnico, kadar je dejanski razred spremenljivke *objekt1* enak razredu spremenljivke *razred* ali pa pripada kateremu od njegovih podrazredov [18].

Pri preverjanju tipov z *instanceof* je pomembno, da najprej preverimo najnižji podrazred, saj lahko v nasprotnem primeru pridemo do napačnih rezultatov. Za primer vzemimo spodnjo programsko kodo 3.3. Spremenljivka *b* ima dejanski tip B, vendar ne smemo pozabiti, da vsebuje spremenljivka tudi tipe vseh nadrazredov. Tako v spodnjem primeru ne bi nikoli prišli do drugega preverjanja *b instanceof B*.

Program 3.3: Primer napačnega preverjanja

```
A b = new B();
if (b instanceof A) {...}.
else if (b instanceof B) {...}
else if ...
```

Pomemben korak pri implementaciji s preverjanjem tipov je pretvarjanje tipov s *(tip)objekt*, kjer tip, ki je bil določen med prevajanjem programa, pretvorimo v podani tip objekta.

Spodnji program 3.4 prikazuje simulacijo mehanizma večkratnega razpošiljanja z zgoraj opisanim pristopom. Kot je razvidno iz primera, je že za preverjanje treh objektov in dveh parametrov metode potrebno napisati precej vrstic kode.

Program 3.4: Primer preverjanja tipov

```
public void visit(Test c1, Test c2) {
    if(c1 instanceof C && c2 instanceof C)
        m((C) c1, (C) c2);
    else if(c1 instanceof C && c2 instanceof B)
        m((C) c1, (B) c2);
    else if(c1 instanceof C && c2 instanceof A)
        m((C) c1, (A) c2);
    else if(c1 instanceof B && c2 instanceof C)
        m((B) c1, (C) c2);
```

```
    else if(c1 instanceof B && c2 instanceof B)
        m((B) c1, (B) c2);
    else if(c1 instanceof B && c2 instanceof A)
        m((B) c1, (A) c2);
    else if(c1 instanceof A && c2 instanceof C)
        m((A) c1, (C) c2);
    else if(c1 instanceof A && c2 instanceof B)
        m((A) c1, (B) c2);
    else if(c1 instanceof A && c2 instanceof A)
        m((A) c1, (A) c2);
    else
        System.out.println("ERR");
}
```

Prednost tega pristopa je enostavnost, vendar kot je že razvidno iz zgornje programske kode, število vrstic kode raste z številom parametrov metode in s številom objektov v hierarhiji. Simulacija večkratnega razpošiljanja z uporabo pristopa s preverjanjem tipov je primerna za uporabo v primeru, ko drugi pristopi niso na voljo.

3.3.2 Uporaba mehanizma odsevnosti

Največja prednost pristopa z uporabo odsevnosti je, da metoda, kjer definiramo izbiranje metode, ni odvisna od števila objektov v hierarhiji. Odločitev, katera metoda bo klicana med izvajanjem programa in določitev dejanskih tipov parametrov, je določena med izvajanjem programa. Mehanizem, ki omogoča te lastnosti, se imenuje odsevnost (*angl. reflection*). Odsevnost je zmožnost, da se program med izvajanjem preverja in pridobi podatke o svoji lastni strukturi ter po možnosti spremeni svoje vedenje in lastno strukturo [24].

Prednosti uporabe odsevnosti:

- Število vrstic kode ne raste s številom konkretnih razredov.

- Za različno število parametrov metode ne potrebujemo različnih metod.

Slabost takšnega pristopa je, da ne omogoča statičnega preverjanja argumentov metode. Če metoda z danimi parametri ne obstaja ali sama podana metoda ne obstaja, nam izpiše napako šele med izvajanjem programa.

Primer implementacije mehanizma odsevnosti je prikazan v spodnjem programu 3.5. Iz testnega razreda kličemo metodo *sprejmi()* iz razreda *VeckratnoRazposiljanje*. Iz metode *sprejmi()* kličemo metodo *poslji()*, kjer s pomočjo mehanizma odsevnosti poizkušamo najti najbolj ustrezno metodo. Če je metodo uspešno najdena, se izvede, v nasprotnem primeru izvajalni sistem vrne napako.

Program 3.5: Primer uporabe odsevnosti

```
class VeckratnoRazposiljanje {
    public Method poslji(Object prejemnik,
                    String ime, Object... parametri) {
        Method m = null;
        int n = parametri.length;
        Class<?>[] razredi = new Class[n];
        int i = 0;
        for (Object o : parametri) {
            razredi[i] = o.getClass();
            i++;
        }
        try {
            method = prejemnik.getClass().
                    getDeclaredMethod(ime, razredi);
            return method;
        } catch (NoSuchMethodException e) {}
        return method;
    }
    public void sprejmi(Object prejemnik, String ime,
                                Object... parametri) {
        Method m = poslji(prejemnik, ime, parametri);
```

```
        try {
            m.invoke(prejemnik, parametri);
        }
        catch (InvocationTargetException e) {
            /** Napaka **/
        }
        catch (IllegalAccessException e) {
            /** Napaka **/
        }
    }
}
```

3.4 Dvojno razpošiljanje

V tem poglavju si bomo pogledali simulacijo dvojnega pošiljanja v programskem jeziku Java. Kot že omenjeno v poglavju 2, je izbira dejanske klicane metode določena glede na dva dinamična tipa: tip objekta prejemnika in tip objekta, ki je podan kot prvi argument.

V nadaljevanju diplomske naloge so predstavljeni naslednji načini simulacije dvojnega razpošiljanja:

- načrtovalski vzorec obiskovalec,
- preverjanje tipov ter
- vzorec načrtovanja odsevnosti.

Tipičen primer uporabe načrtovalskega vzorca obiskovalca je implementacija abstraktnega sintaksnega drevesa (*angl. abstract syntax tree, AST*). AST ponavadi uporabimo pri pisanju prevajalnikov v fazi preverjanja sintaksne pravilnosti programske kode. Tehnike so razložene z enostavnim primerom uporabe AST, kjer naše drevo zajema deklaracijo spremenljivk in aritmetični operaciji seštevanje in dodelitev.
Razredna hierarhija je naslednja (program 3.6):

Program 3.6: Razredna hierarhija *Obisk*

```
interface Obisk {}
    abstract class Izraz implements Obisk {}
        class AritmeticniIzraz extends Izraz {}
            class Sestevanje extends AritmeticniIzraz {}
            class Dodelitev extends AritmeticniIzraz {}
        class Stevilo extends Izraz{}
        class Spremenljivka extends Izraz {}
```

Nad objektno hierarhijo je implementirana operacija *Izpis*, ki izpiše račun in rezultat (razredna hierarhija operacij nad hierarhijo *Obisk* je prikazana v programu 3.7).

Program 3.7: Razredna hierarhija *Obiskovalec*

```
interface Obiskovalec {}
    abstract class ObiskovalecImpl implements
                                    Obiskovalec {}
        class Izpis extends ObiskovalecImpl {}
```

3.4.1 Načrtovalski vzorec obiskovalec

Najpogosteje zasledimo simulacijo dvojnega razpošiljanja z implementacijo načrtovalskega vzorca obiskovalec (*angl. visitor design pattern*).

Načrtovalski vzorec obiskovalec je eden izmed vzorcev vedenjskega načrtovanja (*angl. behavioral design pattern*). Uporabljamo ga za implementacijo operacij nad vsemi elementi objektne strukture. Njegova glavna prednost je omogočanje ločitve operacij od objektne strukture, brez potrebe po preverjanju in pretvarjanju tipov (*angl. type casting*), saj definiramo operacije v ločenem razredu, ki ga imenujemo razred obiskovalec (*angl. visitor*). Če želimo definirati novo operacijo, moramo kreirati nov podrazred razreda obiskovalec [10], [12], [19] in [27].

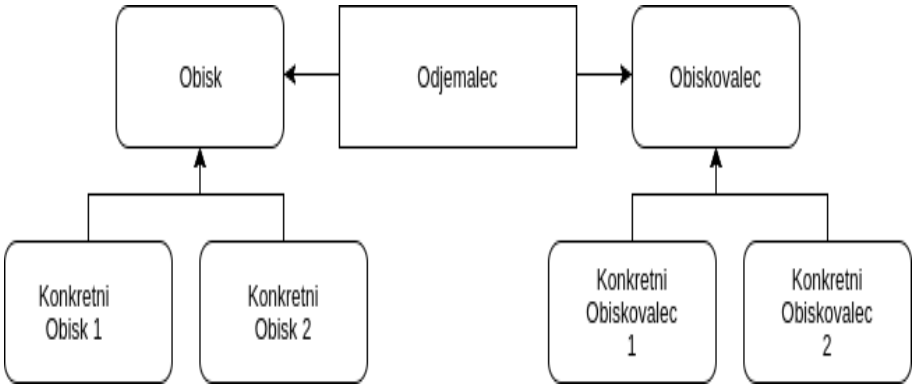
Spodaj podajamo opis in sliko (slika 3.1) glavnih gradnikov načrtovalskega vzorca obiskovalec.

Obisk (*angl. visitable*) Implementacija načrtovalskega vzorca obiskovalca zahteva dva glavna vmesnika *Obisk* in *Obiskovalec*. Prvi vmesnik je vmesnik *Obisk*, kjer deklariramo metodo **sprejmi()** (*angl. accept*), ki sprejme za argument objekt obiskovalec. Pomembno je, da vmesnik implementiramo za razrede, za katere želimo, da so obiskani. To je tudi vstopna točka, ki objektu obiskovalcu omogoča, da obišče objekt.

KonkretniObisk (*angl. concretevisitable*) To so razredi, ki predstavljajo hierarhijo *Obisk*. Vsak razred implementira metodo **sprejmi()** (podedovana od vmesnika *Obisk*), kjer kličemo metodo **obisci()** (*ang. visit*).

Obiskovalec (*angl. visitor*) Vmesnik, kjer deklariramo metodo **obisci()** za vse razrede hierarhije *Obisk*.

KonkretenObiskovalec (*angl. concretevisitor*) Implementira vmesnik *Obiskovalec*, posledično moramo implementirati vse metode **obisci()**, kjer vsaka metoda implementira operacijo nad objektno strukturo. Vsak *KonkretenObiskovalec* predstavlja svojo operacijo.



Slika 3.1: Glavni gradniki vzorca.

Implementacija

Definiramo vmesnik *Obisk* (program 3.8).

Program 3.8: Obisk.java

```
interface Obisk {
    void sprejmi(Obiskovalec obiskovalec);
}
```

Prijaznejši pristop k implementaciji vzorca je, da najprej ustvarimo abstraktni nadrazred 3.9, ki implementira vmesnik *Obisk*. S tem se izognemo implementaciji vmesnika za vsak konkretni razred *Obisk*.

Program 3.9: Izraz.java

```
abstract class Izraz implements Obisk {
    public abstract void sprejmi(Obiskovalec
                                    obiskovalec);
}
```

Naslednja razredna hierarhija 3.10 razširja abstraktni razred *Izraz* in predstavlja *KonkretniObisk*.

Program 3.10: Ostali razredi hierarhije

```
class AritmeticniIzraz extends Izraz {
    public final Izraz prvi;
    public final Izraz drugi;
    public AritmeticniIzraz(Izraz prvi, Izraz drugi)
    {
        this.prvi = prvi;
        this.drugi = drugi;
    }
    public Izraz dobiPrvega() {
        return this.prvi;
    }
    public Izraz dobiDrugega() {
```

```
        return this.drugi;
    }
    public void sprejmi(Obiskovalec obiskovalec) {
        obiskovalec.obisci(this);
    }
}
class Sestevanje extends AritmeticniIzraz {
    public Sestevanje(Izraz prvi, Izraz drugi) {
        super(prvi, drugi);
    }
    public void sprejmi(Obiskovalec obiskovalec) {
        obiskovalec.obisci(this);
    }
}
class Stevilka extends Izraz {
    public final int stevilka;
    public Stevilka(int stevilka) {
        this.stevilka = stevilka;
    }
    public int dobiStevilko() {
        return this.stevilka;
    }
    public void sprejmi(Obiskovalec obiskovalec) {
        obiskovalec.obisci(this);
    }
}
```

Definiramo drugi pomembni vmesnik *Obiskovalec* (program 3.11).

Program 3.11: Obiskovalec.java

```
interface Obiskovalec {
    void obisci(AritmenticniIzraz izraz);
    void obisci(Sestevanje plus);
    void obisci(Stevilka stevilka);
```

```
}
```

V *KonkretenObiskovalec* kreiramo operacije, ki jih želimo izvesti nad zgornjo hierarhijo razredov. V našem primeru želimo izračunati in izpisati vsoto števil (program 3.12).

Program 3.12: Ostali razredi hierarhije *Obiskovalec*

```
abstract class ObiskovalecImpl implements Visitor {
    public abstract void obisci(AritmenticniIzraz izraz);
    public abstract void obisci(Sestevanje add);
    public abstract void obisci(Number number);
}
class Izpis extends ObiskovalecImpl {
    protected int rezultat;
    public int dobiRezultat() {
        return rezultat;
    }
    public void obisci(AritmenticniIzraz izraz) {
        izraz.prvi.sprejmi(this);
        izraz.drugi.sprejmi(this);
    }
    public void obisci(Sestevanje plus) {
        plus.prvi.sprejmi(this);
        int n = rezultat;
        System.out.print("␣+␣");
        plus.drugi.sprejmi(this);
        rezultat = n + rezultat;
    }
    public void obisci(Stevilka stevilka) {
        rezultat = stevilka.dobiStevilko();
        System.out.print(stevilka.stevilka);
    }
}
```

V testnem razredu 3.13 kličemo metodo *sprejmi()*:

Program 3.13: Testni razred

```
public class VisiorDesignPattern {
    public static void main(String[] args) {
        Obisk izraz =
            new Sestevanje(new Stevilka(2),
                new Sestevanje(new Stevilka(3),
                    new Sestevanje(new Stevilka(5),
                        new Stevilka(7))));
        Obiskovalec izpis = new Izpis();
        izraz.sprejmi(izpis);
    }
}
```

Tip spremenljivke *izraz* je med prevajanjem programa *Obisk*, med izvajanjem programa pa *Sestevanje*. Enako velja za spremenljivko *izpis*, med prevajanjem programa je tipa *Obiskovalec*, nato pa *Izpis*.

Poglejmo si simulacijo dvojnega pošiljanja: začetek operacije se začne s klicem *izraz.sprejmi(izpis)*. Java podpira mehanizem enojnega pošiljanja, zato je dejanska klicana metoda, metoda razreda *Sestevanje*, dejanski tipi parametrov metode ne pridejo v upoštev.

Naslednji klic je klic metode *obiskovalec.obisci(this)*. Tukaj se izvrši naslednje enojno razpošiljanje (kar predstavlja tudi drugo razpošiljanje), kar povzroči, da je dejanska klicana metoda metoda v razredu *Izpis* (dejanski tip objekta prejemnika, objekta obiskovalec je *Izpis*).

Slabost tega pristopa je, da ko dodamo nov objekt k objektni hierarhiji *Obisk*, moramo spremeniti vmesnik *Obiskovalec* in posledično tudi vse razrede, ki implementirajo *Obiskovalca*. V razredih moramo implementirati novo metodo *sprejmi()*, ki implementira operacijo nad novim objektom [19].

Dodajmo k objektni hierarhiji *Obisk* objekta *Dodelitev* in *Spremenljivka*. Kot lahko vidimo iz spodnjega primera 3.14, hierarhijo *Obisk* enostavno razširimo.

Program 3.14: Razširitev hierarhije *Obisk*

```
class Dodelitev extends AritmericniIzraz {
    public Dodelitev(Izraz prvi, Izraz drugi) {
        super(prvi, drugi);
    }
    public void sprejmi(Obiskovalec obiskovalec) {
        obiskovalec.obisci(this);
    }
}
class Spremenljivka extends Izraz {
    public int vrednost;
    public final String ime;
    public Spremenljivka(String ime) {
        this.ime = ime;
    }
    public void setVrednost(int vrednost) {
        this.vrednost = vrednost;
    }
    public int getVrednost() {
        return this.vrednost;
    }
    public String getIme() {
        return this.ime;
    }
    public void sprejmi(Obiskovalec obiskovalec) {
        obiskovalec.obisci(this);
    }
}
```

Več popravljanja in dodajanja pride pri implementacij operacij nad objektno hierarhijo. Najprej moramo v vmesniku *Obiskovalec* (program 3.15) deklarirati dve novi metodi *obisci()*. Prva vsebuje parameter objekt *Dodelitev*, druga pa objekt *Spremenljivka*.

Program 3.15: Razširitev hierarhije *Obiskovalec*

```
interface Obiskovalec {
    /**Enako kot zgoraj**/
    void obisci(Dodelitev dodelitev);
    void obisci(Spremenljivka spremenljivka);
}
```

Posledično dodamo v razred *Izpis* novi dve metodi in implementiramo operacije (program 3.16).

Program 3.16: Razširitev razreda *Izpis*

```
public void obisci(Dodelitev dodelitev) {
    dodelitev.prvi.sprejmi(this);
    System.out.print("␣=␣");
    dodelitev.drugi.sprejmi(this);
    System.out.println();
    if(dodelitev.prvi instanceof Spremenljivka)
        ((Spremenljivka)dodelitev.prvi).
                        setVrednost(rezultat);
}
public void obisci(Spremenljivka spremenljivka) {
    rezultat = spremenljivka.getVrednost();
    System.out.print(spremenljivka.getIme());
}
```

3.4.2 Preverjanje tipov

Prednost takšnega pristopa je, da lahko popolnoma ločimo operacije od objektne hierarhije, namreč ni nam potrebno dodajati nobenih dodatnih parametrov ali metod v objektno strukturo. Posledično je naš vmesnik *Obiskovalec* lahko prazen (program 3.17) [19].

Program 3.17: Razširitev razreda *Izpis*

```
interface Obiskovalec {}
```

Naš cilj je, da preverjanje tipov implementiramo samo enkrat in za vsako operacijo, ki jo želimo izvesti nad objektno hierarhijo, kreiramo nov razred. Torej enako kot pri implementaciji vzorca načrtovanja obiskovalca, kreiramo vmesnik *Obiskovalec*, ki deklarira *obisci()* metode za vse razrede v hierarhiji razredov. Abstraktnemu razredu *ObiskovalecImpl* dodamo metodo *obisci()*. Ta metoda je na voljo vsem objektom strukture (program 3.18).

Program 3.18: ObiskovalecImpl.java

```
abstract class ObiskovalecImpl implements Obiskovalec {
    public void obisci(Izraz izraz) {
        if (izraz instanceof Sestevanje)
            obisci((Sestevanje) izraz);
        else if (izraz instanceof Stevilka)
            obisci((Stevilka) izraz);
        else if (izraz instanceof AritmeticniIzraz)
            obisci((AritmeticniIzraz) izraz);
        else
            //NAPAKA
    }
    public abstract void obisci(AritmeticniIzraz izraz);
    public abstract void obisci(Sestevanje plus);
    public abstract void obisci(Stevilka stevilka);
}
```

Sedaj lahko iz testnega programa 3.19 kličemo *obisci()* metodo in s pomočjo preverjanja tipov (*instanceof*) obiščemo pravo *obisci()* metodo. Pomembna je uporaba pretvarjanja tipov *(tip)objekt*, da zagotovimo klicanje prave metode.

Program 3.19: Testni program

```
Izraz izraz = new AritmeticniIzraz(...);
ObiskovalecImpl izpis = new Izpis();
izpis.obisci(izraz);
```

Razred, ki predstavlja operacijo nad razredno strukturo, ima podobno strukturo kot tisti pri implementaciji načrtovalskega vzorca obiskovalec, s tem da namesto klicanja *sprejmi()* metode, sedaj kličemo *obisci()* metodo. Primer metode *obisci()* za objekt *Sestevanje* je prikazan spodaj 3.20.

Program 3.20: primer metode *obisci()* za objekt *Sestevanje*

```
public void obisci(Sestevanje plus) {
    obisci(plus.prvi);
    int n = rezultat;
    obisci(plus.drugi);
    rezultat = n + rezultat;
}
```

3.4.3 Načrtovalski vzorec odsevnost

Problem se pojavi pri načrtovalskem vzorcu obiskovalec, ko želimo dodati nov razred *Obiskovalec*. Namreč v vmesnik *Obiskovalec* je potrebno dodati novo metodo *obisci()* in nato to metodo implementirati v vseh razredih *Obiskovalec*.

V tem poglavju si bomo pogledali fleksibilen pristop simulacije dvojnega pošiljanja s pomočjo načrtovalskega vzorca odsevnost (*angl. reflective design pattern*). Takšen pristop lahko implementiramo v programskih jezikih, ki podpirajo mehanizem odsevnosti, kot na primer java. S pomočjo mehanizma odsevnosti program pregleda (preišče) sam sebe.

Uporabimo ga, kadar naša objektna struktura vsebuje veliko objektov z različnimi vmesniki, mi pa želimo izvajati operacije nad to objektno strukturo. Operacije so lahko različne, nepovezane.

Še posebej je uporaben kadar se naša objektna struktura pogosto spreminja, saj nam mi potrebno ponovno prevajati obstoječih razredov in spreminjati vmesnikov.

Uporaba vpogleda je časovno zahtevna, tako da v sistemih, kjer je pomembna učinkovitost in zmogljivost sistema, ni primerna za uporabo [7], [17].

Komponente, ki jih potrebujemo pri implementaciji načrtovalskega vzorca odsevnost, so podobne komponentam načrtovalskega vzorca obiskovalec.

Obiskovalec Enako kot pri načrtovalskem vzorcu obiskovalec, je ena izmed glavnih komponent vmesnik *Obiskovalec* (lahko tudi abstraktni razred). *Obiskovalec* je glavni, osnovni razred hierarhije. Vsi konkretni razredi *Obiskovalec* izhajajo iz njega. V njem definiramo metodo *obisci()*, ki jo podedujejo konkretni razredi *Obiskovalec*. V testnem razredu pokličemo metodo *obisci()* in pričnemo z izvajanjem operacije nad objektno strukturo. Metoda *obisci()* sprejme za parameter objekt *Obisk*, kjer s pomočjo vgrajenega mehanizma odseva poiščemo najbolj ustrezno metodo *obisci()* za podan objekt.

KonkretenObiskovalec V konkretnem obiskovalcu implementiramo metode *obisci()* za vsak konkreten element. Metode *obisci()* deklariramo kot *protected*, da niso vidne izven sistema.

Obisk Vmesnik *Obisk* implementiramo za vse razrede, za katere želimo, da so obiskani. Pri implementaciji načrtovalskega vzorca odsevnost je vmesnik lahko prazen (ne potrebujemo metode *sprejmi()*). Obiskovalcu zagotavlja dinamične tipe objektov.

Element Pravo tako smo tudi tukaj izbrali abstraktni razred, ki predstavlja nadrazred vseh konkretnih razredov *Element*. Razred *Element* (v našem primeru *Izraz*) implementira vmesnik *Obisk*. Vsi konkretni razredi *Element* izvirajo iz glavnega razreda *Element*.

KonkretenElement Konkretni razredi *Element* in sam razred *Element* sestavljajo razredno hierarhijo *Element*. *KonkretenElement* izvira iz razreda *Element*.

Obiskovalec s pomočjo vpogleda pridobi informacije o *KonkretenElement* in najde najbolj ustrezno metodo *obisci()*. Iskanje se začne v *KonkretenObiskovalec* in potem se nadaljuje po hierarhiji navzgor, do korena. Če je metoda

najdena, se izvede, v nasprotnem primeru predvidevamo, da je metoda definirana v nadrazredu *KonkretenElement*. Proces se ponovi za vse nadrazrede. V primeru, ko pregledamo vse nadrazrede in prave metode *obisci()* še vedno ne najdemo, izvajalni sistem vrne napako [7], [17].

Glavne prednosti uporabe odsevnosti

- Enostavno dodajanje nove operacije nad objektno strukturo, namreč obstoječe programske kode ni potrebno spreminjati, saj dodamo novo operacijo kot nov podrazred hierarhije *Obiskovalec*.
- Objektna struktura je popolnoma neodvisna od hierarhije *Obiskovalec*, saj izvede ponovno enojno pošiljanje v razredu *Obiskovalec*.
- Samo metoda *obisci()* je vidna izven hierarhije *Obiskovalec*, tako da v testnem razredu pokličemo to metodo in pričnemo z izvajanjem operacije nad objektno hierarhijo.

Slabosti

- V programskem jeziku, kot na primer C++, ne moremo implementirati načrtovalskega vzorca odsevnosti, saj jezik ne podpira mehanizma odsevnosti.
- Uporaba vzorca odsevnosti ni primerna v časovno kritičnih sistemih, saj zmanjša učinkovitost in zmogljivost sistema.
- Ime metode *obisci()* mora biti enako v vseh razredih hierarhije *Obiskovalec*.

Implementacija

V abstraktnem razredu *ObiskovalecImpl* deklariramo metodo *obisci()* z argumentom objekta *Obisk* (program 3.21). Iz te metode kličemo metodo *getMethod(...)*, kjer nam nato ta metoda vrne najbolj ustrezno metodo z uporabo odsevnosti.

Program 3.21: Primer metode *obisci()* v razredu *ObiskovalecImpl*

```
public void obisci(Obisk obisk) {
    Method method = getMethod(obisk);
    try {
        method.invoke(this, obisk);
    }
    catch (InvocationTargetException e) {
        /** NAPAKA **/
    }
    catch (IllegalAccessException e) {
        /** NAPAKA **/
    }
}
```

Metodo *getMethod()* lahko implementiramo s preiskovanjem v širino in preiskovanjem v globino.

Preiskovanje v širino

Preiskovanje v širino (*angl. breadth search first, BFS*) pomeni, da najprej pregledamo hierarhijo *Obisk* in se premikamo še po hierarhiji *Obiskovalec*.

Preiskovanje v širino implementiramo z ugnezdeno zanko (program 3.22). V notranji zanki najprej poskušamo najti najbolj ustrezno metodo *obisci()* s podanim parametrom. Če za dani parameter ne obstaja ustrezna metoda, se premaknemo na nadrazred parametra. Postopek ponavljamo dokler ne pridemo do korena objektne strukture. V zunanji zanki se premikamo po hierarhiji *Obiskovalec*. Tako da v primeru, ko smo v notranji zanki prišli do korena, se v zunanji zanki premaknemo na nadrazred podanega *Obiskovalca* in tam poskušamo najti najbolj ustrezno metodo. Postopek ponavljamo, dokler ne pridemo do korena hierarhije *Obiskovalec*. Če metode še vedno nismo našli, vrnemo napako.

Program 3.22: Primer preiskovanja v širino

```
protected Method getMethod(Obisk obisk) {
    Method method = null;
    Class obiskovalecClass = getClass();
    Class ObiskClass = obisk.getClass();
    while(obiskovalecClass != null &&
                obiskovalecClass != Obiskovalec.class){
        ObiskClass = obisk.getClass();
        while(ObiskClass != null &&
                            ObiskClass != Obisk.class){
            try {
                method = obiskovalecClass.
                    getDeclaredMethod("obisci",
                            new Class[]{ObiskClass});
            } catch (NoSuchMethodException e) {}
            ObiskClass = ObiskClass.getSuperclass();
        }
        obiskovalecClass =
                    obiskovalecClass.getSuperclass();
    }
    return method;
}
```

Preiskovanje v širino je primerno kadar imamo veliko objektno hierarhijo *Obisk* in lahko za kakšen razred in njegov podrazred definiramo enako operacijo.

Preiskovanje v globino

Pri preiskovanju v globino (*angl. depth search first, DFS*) najprej pregledamo hierarhijo *Obiskovalec*, kjer poskušamo najti najbolj ustrezno metodo. Če je ne najdemo, se premikamo še po hierarhiji *Obisk* navzgor. Implementacija preiskovanja v globino je prikazana v programu 3.23.

Program 3.23: Primer preiskovanja v globino

```
protected Method getMethod(Visitable visitable) {
    Method method = null;
    Class obiskovalecClass = getClass();
    Class ObiskClass = visitable.getClass();
    while(ObiskClass != null &&
                    ObiskClass != Obisk.class){
        obiskovalecClass = getClass();
        while(obiskovalecClass != null &&
                obiskovalecClass != Obiskovalec.class){
            try {
                method = obiskovalecClass.
                            getDeclaredMethod("obisci",
                               new Class[]{ObiskClass});
                return method;
            } catch (NoSuchMethodException e) {}
            obiskovalecClass =
                    obiskovalecClass.getSuperclass();
        }
        ObiskClass = ObiskClass.getSuperclass();
    }
    return method;
}
```

Preiskovanje v globino je primernejše v primeru, ko razširjamo hierarhijo *Obiskovalec*. Sledi primer, kako lahko na enostaven način, brez spreminjanja obstoječih razredov dodamo nove operacije:

- Imamo objektno strukturo *Obisk* in *Obiskovalec*

```
interface Obisk {}
    abstract class Izraz {}
        class AritmeticniIzraz {}
            class Sestevanje {}
        class Stevilo {}
```

```
interface Obiskovalec {}
    abstract class ObiskovalecImpl {}
        class Izpis {}
```

- K objektni strukturi dodamo nova objekta *Dodelitev* in *Spremenljivka*. *Dodelitev* razširja razred *AritmeticniIzraz* in razred *Spremenljivka* razširja *Izraz*.
- Ker ne želimo spreminjati obstoječih razredov v hierarhiji *Obiskovalec*, dodamo nov razred *podIzpis*, ki razširja razred *Izpis*.
- V razredu *podIzpis* deklariramo metodi *obisci()*, za nova objekta *Dodelitev* in *Spremenljivka*.
- Iz testnega razreda nato kličemo metodo *obisci()* za novi razred *podIzpis* *podIzpis.obisci(...)*.

Poglavje 4

Eksperimentalno ovrednotenje

V tem razdelku je izmenjana hitrost delovanja zgoraj predstavljenih pristopov. Najprej je izmerjena hitrost izvajanja dvojega razpošiljanj, nato pa še izvajanja večkratnega razpošiljanja, kjer primerjamo hitrost razpošiljanja do štirih parametrov metode.

Vsi testi so bili izvedeni na operacijskem sistemu Ubuntu 17.04, na računalniku s procesorjem Intel Core i5-5200U s štirimi jedri in z delovnim pomnilnikom velikost 12 GB.

Hitrost izvajanja programov je merjena z ukazom *System.nanoTime()*, kjer dobimo čas od zagona javanskega virtualnega sistema v nanosekundah. Ta čas moramo nato deliti z milijonom, da dobimo čas v milisekundah.

Za vsak testni primer je bilo izmerjenih deset meritev čas izmerjen desetkrat. Iz meritev je bila nato izračuna aritmetična sredina in standardni odklon. Aritmetično sredino izračunamo kot seštevek desetih meritev, ki ga nato delimo z deset. Standardni odklon prestavlja razpršenost rezultatov.

4.1 Dvojno razpošiljanje

Dvojno razpošiljanje je bilo testirano z enostavnim AST (razredna hierarhija je razvidna iz programa 3.6) iz poglavja 3.4, kjer so bili za potrebe testiranja izdelani različni testi, ki si jih bomo pogledali v naseljenih razdelkih.

4.1.1 Vsota številk

V prvem testu merimo hitrost izračuna, kjer seštejemo prvih štirinajst praštevil (prikazano spodaj 4.1). Da pa račun ne bo prelahek, najprej nastavimo štirim spremenljivkam vrednosti in nato te spremenljivke vključimo v izračun.

Program 4.1: Vsota praštevil

```
x = 7
y = 17
z = 37
w = 47
2 + 3 + 5 + x + 11 + 13 + y + 19 + 23 +
                        29 + 31 + z + 41 + w = ?
```

Spodnja tabela 4.1 prikazuje povprečne čase in standardni odklon meritev desetih meritev. Razvidno je, da je najhitrejši pristop s preverjanjem tipov, vidimo pa tudi, da je načrtovalski vzorec obiskovalec takoj za njim. Najpočasnejša pristopa sta pristopa z uporabo odsevnosti, saj je mehanizem odsevnosti časovno zahteven.

Tehnika	Čas [ms]	Standardni odklon
Preverjanje tipov	0.05	0.01
Obiskovalec	0.05	0.01
Odsevnost BFS	0.87	0.07
Odsevnost DFS	0.99	0.21

Tabela 4.1: Čas izvajanja različnih tehnik za primer *Vsota praštevil*

4.1.2 Vsota spremenljivk

Naslednji test testira učinkovitost seštevanja vrednosti spremenljivk. Rezultat testa je vsota šestindvajsetih spremenljivk. Vsaka spremenljivka ima drugačno vrednost. Naš račun je naslednji (4.2):

Program 4.2: Vsota spremenljivk

```
a = 1
b = 2
...
v = 21
z = 22
q = 23
x = 24
y = 25
w = 26
a + b + c + d + e + f + g + h + i + j + k + l + m +
            n + o + p + r + s + t + u + v + z + q +
                                        x + y + w = ?
```

Iz tabele 4.2 lahko razberemo, da je pri tem testu najhitrejši pristop z uporabo načrtovalskega vzorca obiskovalec, saj je naše število objektov naraslo. Zato pristop s preverjanjem tipov potrebuje več časa, da lahko preveri vse objekte. To je tudi posledica počasnejšega izvajanja pristopov z uporabo odsevnost. Vidimo lahko tudi, da je BFS precej hitrejši od DFS, saj s tem, ko imamo veliko hierarhijo *Obisk* in majhno hierarhijo *Obiskovalec*, lahko zunanjo zanko pri BFS praktično zanemarimo (pri BFS se najprej sprehodimo po hierarhijo *Obisk* (notranja zanka), v zunanji zanki pa se sprehodimo po hierarhiji *Obiskovalec*, glej program 3.22).

Tehnika	Čas [ms]	Standardni odklon
Obiskovalec	0.09	0.015
Preverjanje tipov	0.12	0.07
Odsevnost BFS	0.74	1.09
Odsevnost DFS	8.05	0.90

Tabela 4.2: Čas izvajanja različnih tehnik za primer *Vsota spremenljivk*

4.1.3 Pristop z uporabo odsevnosti

V tem razdelku si bomo pogledali, kdaj je primernejše uporabiti preiskovanje v globino in kdaj preiskovanje v širino, kadar želimo simulirati dvojno razpošiljanje z uporabo mehanizma odsevnosti.

Razredno hierarhijo *Obiskovalec* iz razredne hierarhije predstavljeno v poglavju 3.4 (program 3.7), ki predstavljajo operacije nad objektno hierarhijo(program 3.6), razširimo tako, da je vsaka operacija nad enim objektom predstavljena kot svoj razred (program 4.3).

Program 4.3: Sprememba hierarhije *Obiskovalec*

```
class IzrazImpl extends ObiskovalecImpl {
    //obdelaj aritmeticni izraz
    public void obisci(AritmenticniIzraz izraz) {
        obisci(izraz.prvi);
        obisci(izraz.drugi);
    }
}
class PlusImpl extends IzrazImpl {
    //sestej dva izraza
}
class StevilkaImpl extends PlusImpl {
    //obdelaj stevilko
}
```

```
class DodeliImpl extends StevilkaImpl {
    //dodeli vrednost spremenljivki
}
class Spremenljivka extends DodeliImpl {
    //obdelaj spremenljivko
}
```

Sedaj imamo razširjeno hierarhijo *Obiskovalec*. Naša dva pristopa z uporabo mehanizma odsevnosti se bosta morala sprehoditi po vseh razredih in poiskati pravo metodo.

Tabela 4.3 prikazuje rezultate testa z vsoto spremenljivk, tabela 4.4 pa rezultate testa z vsoto praštevil. Čas izvajanja je počasnejši za oba preiskovanja, kar je posledica razširitve hierarhije *Obiskovalec*. Namreč metodo, ki jo bo izvajalni sistem dejansko klical, moramo iskati v več razredih. Iz tabel je razvidno, da je v našem primeru precej bolje uporabiti preiskovanje v globino, saj je hitrejše kot preiskovanje v širino.

Tehnika	Čas [ms]	Standardni odklon
Odsevnost BFS	24.06	2.04
Odsevnost DFS	13.69	1.20

Tabela 4.3: Čas izvajanja testa Vsota spremenljivk z razširjeno hierarhijo *Obiskovalec*

Tehnika	Čas [ms]	Standardni odklon
Odsevnost BFS	6.12	1.11
Odsevnost DFS	2.82	0.82

Tabela 4.4: Čas izvajanja testa Vsota spremenljivk z razširjena hierarhija *Obiskovalec*

4.2 Večkratno razpošiljanje

Za merjenje hitrosti izvajanja večkratnega razpošiljanja sta bila uporabljena naslednja dva pristopa:

- odsevnost ter
- preverjanje tipov.

Za vsak pristop sem uporabila štiri različne testne scenarije, kjer je dejanska klicana metoda izbrana glede na:

- en parameter metode,
- dva parametra metode,
- tri parametre metode,
- štiri parametre metode.

Za potrebe testiranja sem uporabila zgornjo razredno hierarhijo (program 3.2), kjer imamo razrede A, B, C, D (A razširja B, B razširja C in C razširja D).

V testu, kjer je izbira metode določena glede na en parameter metode, izbiramo med štirimi metodami, kjer vsaka sprejme en zgornji razred (primer program 4.4).

Program 4.4: Primer metod z enim parametrom

```
public String m(A a) {
    return "A";
}
public String m(B b) {
    return "B";
}
public String m(C c) {
    return "C";
}
```

```
public String m(D d) {
    return "D";
}
```

Za dva parametra je napisanih šestnajst metod, to je vse možne variacije (s ponavljanjem) dveh izmed zgornjih štirih razredov. Primeri metod z dvema parametroma so naslednji 4.5:

Program 4.5: Primer metod z dvema parametroma

```
public String m(A a, A a2) {
    return "AA";
}
public String m(A a, B b) {
    return "AB";
}
...
public String m(D d, D d2) {
    return "DD";
}
```

S tremi parametrih izvajalni sistem izbira med 64 metodami, primer nekaj metod je prikazan spodaj 4.6.

Program 4.6: Primer metod z tremi parametri

```
public String m(A a, A a2, A a3) {
        return "AAA";
}
...
public String m(D d, D d2, D d3) {
    return "DDD";
}
```

In za testni scenarij, kjer je izbira metode določena glede na štiri parametre, izbiramo med 256 metodami. Primer metode je prikazan v programu 4.7.

Program 4.7: Primer metod s štirimi parametri

```
public String m(A a, A a2, A a3, A a4) {
    return "AAAA";
}
...
public String m(D a, D a2, D a3, D a4) {
    return "DDDD";
}
```

V testnem razredu kreiramo instance razredov, kjer so vsi statičnega tipa nadrazreda A. Nato pa v zanki tisočkrat kličemo štiri metode, ki vsebujejo različne tipe parametrov. Naslednji dve tabeli prikazujeta rezultate meritev.

Iz tabele 4.5, kjer so predstavljene meritve pristopa z uporabo mehanizma odsevnosti, lahko razberemo, da čas izvajanja programa ne narašča z naraščanjem parametrov in številom metod. Temveč se čas zmanjša in počasi ustali (razvidno tudi iz grafa 4.2).

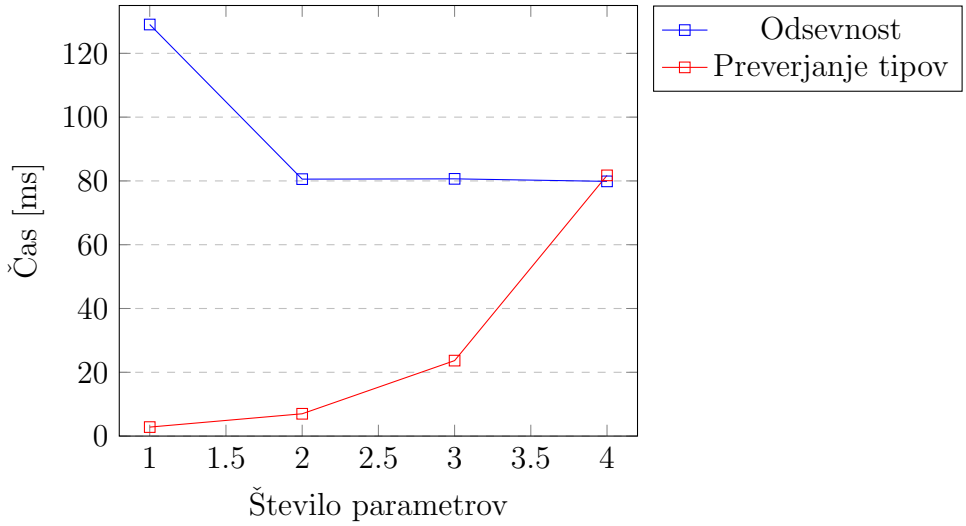
Enako kot meritve pristopa z uporabo mehanizma odsevnosti, so v tabeli 4.5 predstavljene tudi meritve pristopa s preverjanjem tipov. Čas izvajanja programa hitro narašča s povečevanjem števila parametrov metode in s številom metod. Pri preverjanju tipov ne narašča le čas izvajanja, temveč tudi število vrstic kode. Za štiri različne tipe objektov je bilo potrebno napisati več kot 200 preverjanj (seveda, če preverimo čisto vse možne variacije (s ponavljanjem)).

Za lažjo predstavo je bil izdelan tudi graf, ki ponazarja čas izvajanja v odvisnosti z naraščanjem parametrov. Z rdečo so prikazane meritve pristopa s preverjanjem tipov, z modro pa pristop z odsevnostjo.

	Preverjanje tipov		Odsevnost	
Št. parametrov	Čas [ms]	standardni odklon	Čas [ms]	standardni odklon
1	2.82	0.80	129.04	3.75
2	6.99	1.78	80.56	3.24
3	23.66	3.04	80.65	4.59
4	81.74	5.41	79.86	3.23

Tabela 4.5: Rezultati meritev večkratnega razpošiljanja

Primerjava časa izvajanja večkratnega razpošiljanja



Poglavje 5

Zaključek

V diplomski nalogi smo predstavili različne pristope, kako lahko na lažji pa tudi na zahtevnejši način simuliramo večkratno in dvojno razpošiljanje v programskem jeziku, kot je na primer java, ki podpira samo enojno razpošiljanje.

Simulacije večkratnega razpošiljanja je primernejša s pristopom uporabe mehanizma odsevnosti, saj število vrstic ne raste s številom razredov v hierarhiji in metodo, kjer z uporabo mehanizma odsevnosti najdemo najbolj ustrezno metodo, implementiramo samo enkrat. Implementacija pristopa s preverjanjem tipov je lažja, vendar se je izkazalo, da ga je primerno implementirati samo v primeru, ko imamo manjšo razredno hierarhijo, saj v tem primeru ne potrebujemo veliko preverjanj tipov.

Za simulacijo dvojega razpošiljanja je najboljše uporabiti načrtovalski vzorec obiskovalec, kadar se naša razreda hierarhija ne spreminja pogosto in ko želimo, da je naš program hiter. V primeru, ko se razredna hierarhija pogosto spreminja ali če dodajamo veliko novih operacij, ki delujejo nad to hierarhijo, je primernejše uporabiti načrtovalski vzorec odsevnost. Seveda, če naš sistem ni časovno kritičen, saj je mehanizem odsevnosti časovno potraten. Najlažja je implementacija s pristopom preverjanja tipov, vendar v primeru, ko imamo veliko število različnih tipov objektov, ni primerna, saj potrebuje veliko število vrstic kode, da lahko preverimo vse tipe objektov.

Literatura

[1] Luca Cardelli. A semantics of multiple inheritance. *Information and computation*, 76(2-3):138–164, 1988.

[2] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. *SIGPLAN Not.*, 35(10):130–145, October 2000.

[3] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, May 2006.

[4] Apple Computer. The Dylan Reference Manual. `https://opendylan.org/books/drm/Language_Overview`. Dostopano: 2017-09-03.

[5] Jeff Dalton. A Brief Guide to CLOS. `http://www.aiai.ed.ac.uk/~jeff/clos-guide.html`. Dostopano: 2017-09-01.

[6] Christopher Dutchyn, Paul Lu, Duane Szafron, Steven Bromling, and Wade Holst. Multi-dispatch in the java virtual machine: Design and implementation. In *Proceedings of the 6th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 6*, COOTS'01, pages 6–6, Berkeley, CA, USA, 2001. USENIX Association.

[7] Rémi Forax, Etienne Duris, and Gilles Roussel. Reflection-based implementation of java extensions: The double-dispatch use-case. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 1409–1413, New York, NY, USA, 2005. ACM.

[8] The Common Lisp Foundation. Common Lisp. https://common-lisp.net/. Dostopano: 2017-09-01.

[9] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[10] Lokesh Gupta. Visitor Design Pattern Example. https://howtodoinjava.com/design-patterns/behavioral/visitor-design-pattern-example-tutorial/. Dostopano: 2017-07-05.

[11] Dylan Hackers. An Introduction to Dylan. https://opendylan.org/documentation/intro-dylan/multiple-dispatch.html. Dostopano: 2017-09-03.

[12] Rohit Joshi. Visitor Design Pattern Example. https://www.javacodegeeks.com/2015/09/visitor-design-pattern.html. Dostopano: 2017-08-16.

[13] Julia. JuliaCon 2017. https://julialang.org/. Dostopano: 2017-06-26.

[14] Julia. The Julia Language. https://julialang.org/. Dostopano: 2017-06-26.

[15] Nick Levine. Fundamentals of CLOS. http://cl-cookbook.sourceforge.net/clos-tutorial/. Dostopano: 2017-09-01.

[16] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[17] Yun Mai and Michel De Champlain. Reflective visitor pattern, 2001.

[18] Uroš Mesojedec and Borut Fabjan. Java 2: temelji programiranja, 2004.

[19] Jurij Mihelič and Igor Rožanc. Techniques for traversal operation on an object structure: a comparison. *Central European Conference on Information and Intelligent Systems*, pages 213–220, 2015.

[20] MultiJava. The MultiJava Project. `http://multijava.sourceforge.net/`. Dostopano: 2017-06-25.

[21] Radu Muschevici. Multiple dispatch in practice. Thesis, Victoria University of Wellington, 2009.

[22] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. *SIGPLAN Not.*, 43(10):563–582, October 2008.

[23] Milan Ojsteršek. e-računlništvo polimorfizem. `http://www.s-sers.mb.edus.si/gradiva/rac/moduli/programske_aplikacije/66_polimorfija/01_datoteka.html`. Dostopano: 2017-08-01.

[24] Francisco Ortin, Jose Quiroga, Jose M. Redondo, and Miguel Garcia. Attaining multiple dispatch in widespread object-oriented languages. 2015.

[25] Chris Riesbeck. The Common Lisp Object System. `https://www.cs.northwestern.edu/academics/courses/325/readings/clos.php`. Dostopano: 2017-09-01.

[26] Asim Anand Sinha and Sophia Drossopoulou. *Multiple Dispatch and Roles in OO Languages: FickleMR*. PhD thesis, 2005.

[27] SourceMaking. Visitor Design Pattern. `https://sourcemaking.com/design_patterns/visitor`. Dostopano: 2017-08-16.